Extending Recommendation Systems with Software Maps

Jonas Trümper Jürgen Döllner
Hasso-Plattner-Institute – University of Potsdam, Germany
{*jonas.truemper|juergen.doellner*}*@hpi.uni-potsdam.de*

***Abstract*—In practice, recommendation systems have evolved as helpful tools to facilitate and optimize software engineering processes. Serving both developers and managers, specific recommendation systems address their individual problems. Yet, in a number of cases complementing them with other techniques can enhance their use and extend their scope. In this paper, we first discuss different perspectives on software-engineering processes and examples of recommendation systems that support representatives of these perspectives. We then identify how select software-map techniques can extend recommendation systems to facilitate decision making by addressing the perspectives' information and communication needs.**

***Keywords*-Decision making; Context; Computer aided analysis; Visualization**

I. Introduction

Today's recommendation systems (RS) for software engineering provide meaningful insights and valuable proposals (e.g., a ranked result list) to their users [1]. Focusing on various sub-domains in software engineering, their application ranges from refactoring suggestions to a manager's dashboard, which, e.g., recommends postponing a product release due to a potentially critical increase in bug count close to the scheduled release date.

Nevertheless, a recommendation naturally implies the crucial question whether one will follow the given proposal. Besides insufficient trust in recommendations [2], it may simply be necessary for users to either understand why a recommendation resulted or – despite a sophisticated recommendation system – to check plausibility of a proposal and its applicability to their current working context. So, evaluation can be conducted along many dimensions and can require additional context information to decide whether to follow a given recommendation. That is, such decision-making process finally transforms a machine-made proposal into a man-made decision.

Moreover, an RS may trigger communication between stakeholders in a software-engineering project to achieve a multi-person decision. An RS, however, typically does not provide a suitable means to communicate about the respective topic, which also is not its purpose. For instance, if an RS indicates to a manager that code quality is decreasing and poses a long-term risk, software architects and developers ultimately have to explain whether that indication is true and eventually convince the manager of their planned solution to this issue.

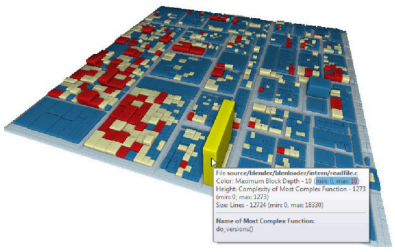

Figure 1. Treemap of a system's static structure (courtesy of Bohnet and Döllner [3]) showing per-file outliers in max. function complexity (→ height), lines of code (→ size), and max. nesting level (→ color).

Software maps provide a powerful supportive means in these cases to either serve information needs or to bridge communication gaps. We use the term 'software maps' for visual techniques that combine thematic information about software development processes (evolution), software quality, structure, and dynamics and display that information in a cartographic manner. Software maps can be used to check upcoming decisions for plausibility (is a decision for a specific alternative reasonable or are there arguments against it?) and to determine whether decisions are overdue (do any facts indicate that a decision should have been made already?). For instance, in Fig. 1 a hierarchical system structure is depicted by nested blocks and multiple software quality attributes are combined as distinct visual attributes. By this, the software map allows for reasoning about potential code quality per module and whether a decision to restructure the most monolithic and complex code entity (large yellow block) should have been made already.

In this paper, we first briefly review different perspectives on software-engineering processes and how they typically relate to each other with respect to communication needs and topics. The main contribution of this paper includes a discussion of selected recommendation systems that aim at supporting representatives of identified perspectives and a proposal how software-map techniques can help making decisions based on the respective system's recommendations.

II. Key Perspectives, Information Needs, and Communication Needs

In many larger software development projects, there are a number of key perspectives on the underlying process and its deliverables that exist regardless of the methodology used

978-1-4673-1759-7/12/$31.00 © 2012 IEEE

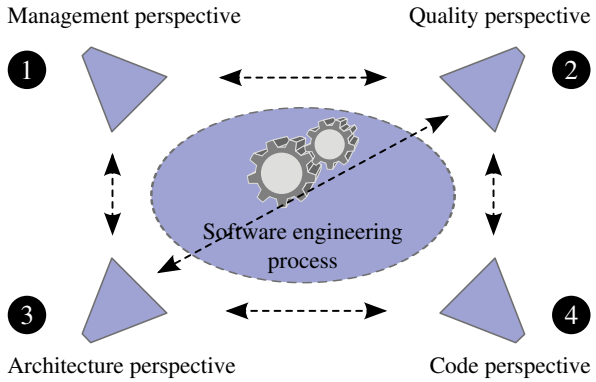

Figure 2. Typical perspectives in larger software engineering projects. Links between perspectives represent communication channels.

for software development. First, these perspectives differ in terms of abstraction level and, second, in information needs and communication needs. Software engineering generally does not focus on such interconnected view on these perspectives and their needs. Thus, we provide a discussion of three of these perspectives (management, quality and architecture) based on observations that we made in large industrial software projects. In the remainder, existence of a need is encoded by $\overline{X}$ and satisfaction of a need by $\underline{X}$.

The management perspective (1 in Fig. 2): Personnel in this perspective are responsible for ongoing work and future directions, and ultimately decide on resources. They are typically non-technical staff and thus it is essential for them to get condensed, high-level information as overview of the situation ($\overline{M1}$). However, during subsequent decision-making processes, they may need on-demand access to additional, detailed information ($\overline{M2}$). Some of this information can be retrieved from update-to-date fact sheets of projects ($\overline{M3}$). In all remaining cases, personnel in this perspective also communicate with other representatives of the management perspective as well as the quality perspective and the architectural perspective ($\overline{M4}$). A representation of the subject information under discussion that is understood by all participating personnel is essential then ($\overline{M5}$).

The quality perspective's (2 in Fig. 2) main objective is to balance internal software quality (e.g., maintainable code) and external software quality (functionality visible to customers). With respect to external software quality, requirements, their representation as tests, and their realization are important subjects when representatives of this perspective communicate with personnel in the management perspective. In addition, they need to constantly determine how well each sub module's functionality is secured by tests and where to invest future testing effort ($\overline{Q1}$). For managing internal software quality, up-to-date information on system structure ($\overline{Q2}$), on internal structure of modules ($\overline{Q3}$), and assessments of maintainability and change impact ($\overline{Q4}$) are important factors [4].

The architecture perspective (3 in Fig. 2): These personnel are in charge of a system's architecture and coordinate development efforts, so they need to be aware of a system's entire structure ($\overline{A1}$) and current development activity ($\overline{A2}$). In addition, working out how to meet user requirements with a software-based solution to implement the 'big picture' [5], they are in contact with both the management perspective as well as the code perspective. Therefore, they constantly need to translate between both worlds, which in turn requires suitable multi-level representations of systems ($\overline{A3}$). Further, their main concerns on a system-wide scope include (future) maintainability and longterm risks. Essential methods in this context include refactoring and restructuring. Both require information on how a system's structure can be changed ($\overline{A4}$), potential change impact [6] ($\overline{A5}$), and change effort ($\overline{A6}$).

By analogy with individual viewing angles in Fig. 2, each perspective has its own view and abstraction of a software engineering process. This reflects in different requirements with respect to needed information and information to be communicated (Table I). As general rule, the more abstract a perspective's view on software-engineering processes, the more multi-dimensional and holistic (i.e., 'strategic') are its information and communication needs. On the contrary, the more detailed the view of a perspective is, the more concrete and analytical (i.e., 'technical') are its needs.

Table I
NEEDS OVERVIEW PER PERSPECTIVE.

Perspective	Id	Need description
Management	M1	Condensed, high-level information
	M2	On-demand context information
	M3	Up-to-date overview fact-sheets
	M4	Discuss with other perspectives
	M5	Representation understood by all parties
Quality	Q1	Info. on where to invest testing effort
	Q2	Up-to-date system structure
	Q3	Up-to-date internal module structure
	Q4	Info. on maintainability and change impact
Architecture	A1	Up-to-date system structure
	A2	Up-to-date information on development activity
	A3	Multi-level representation of the system or facts
	A4	Restructuring possibilities
	A5	Change impact assessments
	A6	Change effort assessments

III. SOFTWARE MAPS AND RECOMMENDATION TECHNIQUES

Recommendation techniques operate on different levels of abstraction in software engineering, including management-level, quality-level and architecture-level. We examine representatives of these three levels that address identified needs. Due to space restrictions, we only discuss a few prominent examples. We then discuss where software maps can be used to facilitate decision-making based on these techniques' recommendations.

A. Techniques: Management Perspective

Techniques for predicting maintenance effort [7, 8] are typically used by management personnel and provide indications for problematic situations in projects ($\underline{M1}$). Yet, important context information is typically omitted, but is often essential to find out whether such indicated problem is really to be considered as severe. For instance, if an RS demands for action due to a significant increase in reported bugs, this does not automatically imply that external software quality has dropped. This increase can as well be due to increased test coverage in areas that were sparsely tested before, but of which management personnel was not aware. Resulting needs thus include an overview of other indicators ($\overline{M3}$), exploration techniques to identify possible correlations between these indicators ($\overline{M2}$) and suitable representations ($\overline{M5}$, $\overline{A3}$) that management personnel can use in discussions with personnel in the perspectives architecture and quality ($\overline{M4}$).

Treemaps [3], as an overview visualization, allow for mapping multiple attributes to hierarchical structures such as a system's static structure or even a system's landscape composed of many software systems ($\underline{M2}$, $\underline{M5}$, $\underline{A3}$). Exploring correlations between multiple dimensions, such as aforementioned bug count *and* test coverage or development activity ($\underline{A2}$), then helps discussing ($\underline{M4}$) and understanding the real causes of recommendations, and thereby supports deciding whether to follow recommendations.

Pereira et al. [9] present an approach to recommend locality-oriented team allocation schemes based on teams' communication requirements, which focuses on software team allocation at global scope ($\underline{M1}$). Communication requirements are derived from module dependencies that teams are assigned to. In particular, when faced with alternative solutions exposing only subtle differences, in-depth analysis of communication requirements with representatives of the architecture perspective is inevitable to form a decision ($\overline{M2}$, $\overline{M4}$, $\overline{M5}$).

Circular Bundleviews [10] (Fig. 3) provide a means to address these needs by enabling analysis of aforementioned dependencies (i.e., communication requirements) in detail ($\underline{M2}$): Hierarchy is mapped to concentric rings, where each hierarchy level is represented by a ring, subdivided into the number of elements at that level. The circle in between these rings is then used to depict other relationships in a bundled manner, which emphasizes hierarchical containment and similar relationships. By this, the number of crossings and resulting visual clutter is largely reduced and relationships to modules with the same parent hierarchy are visually close together. Relationships that cross module boarders – and are thus likely to require cross-team communication – can be clearly discerned from those that stay local within a team's module, which transforms the puzzling technical problem (identify dependencies requiring communication) to a clear

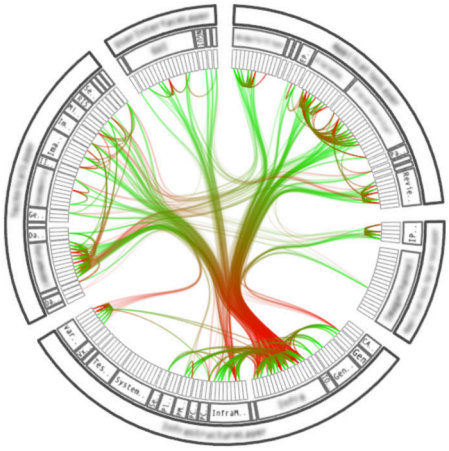

Figure 3. A circular bundleview that depicts hierarchical structure as concentric rings and bundles relationships between nodes in the center (courtesy of Holten [10]).

indicator for all parties ($\underline{M4}$, $\underline{M5}$). Further, non-optimal team allocation in existing projects can be identified, i.e., the map helps find out whether a decision to optimize team allocation should have been made already.

B. Techniques: Quality Perspective

To set test emphasis ($\underline{Q1}$), a prioritization of system modules to test is needed that essentially encodes their criticality. A general approach to this problem is to determine a ranking of complex and frequently changed modules [11]. While these criteria allow for providing a helpful pre-selection, a more fine-grained examination of respective modules will typically follow. For example, locality of relationships plays a role in criticality ($\overline{Q2}$, $\overline{Q3}$), i.e., the chance of a module to cause a system failure correlates with the number of other modules that depend on its correct behavior.

For this RSs, various software-maps can be used for different levels of structural analysis. At high detail level, *DependencyViewer* [12] depicts UML-like compound graphs at package or class level (Java systems) ($\underline{Q3}$). For larger structures, circular bundleviews can be helpful: They visually distinguish rather local relationships (e.g., two submodules of the same parent) from non-local relationships through their hierarchical relationship bundling and remain useful even with thousands of relationships ($\underline{Q2}$).

Approaches to help maintain code correctness and to support maintainability ($\underline{Q4}$) include assistance for API usage [13, 14, 15] and recommendations for changing the invocation context of a method call to reflect the contexts of most other calls to this method [16]. Both scenarios can involve quite complex recommendations, especially in the context of parallel execution which complicates understanding and debugging processes, because there is no direct mapping of source code to runtime behavior. For instance, to decide whether a proposal to change a method invocation context is

applicable to a specific piece of code, the proposed context changes and their impact on program behavior have to be understood beforehand [4] ($\overline{Q2}$, $\overline{Q3}$, $\overline{Q4}$). To support this, call graphs [17] or call timelines [18] can be exploited during proposal. They can provide a visual representation of proposed changes by interleaving current code context with proposed context and highlighting differences and can be used as representation to discuss change impact ($\underline{Q2}$, $\underline{Q3}$, $\underline{Q4}$).

C. Techniques: Architecture Perspective

Hummel et al. [19] present a system that primarily aids architecture personnel both during the design phase of software systems and during later restructuring measures. It recommends similar (object-oriented) designs found in open-source repositories by matching 'footprints' of respective classes (class name and methods) ($\underline{A4}$). Based on the assumptions that self-explaining class names and methods enable us to conclude implemented functionality, this is certainly a well-usable initial selection criterion. Applying a proposed design, however, also implies accepting its associated maintenance effort, with all its benefits and weaknesses. Hence, users likely need to conduct a careful analysis – such as which relationships exist between classes (usage, inheritance, etc.) – before applying any third-party design ($\overline{A5}$, $\overline{A6}$).

Augmented user interfaces of development environments help identifying refactoring opportunities based on code smells [20, 21] ($\underline{A4}$). Each refactoring suggestion is explained to show the reasoning behind. However, the more complex the subject code is, the more complex it is to envision what a refactored solution will look like and what change impact is implied ($\overline{A5}$, $\overline{A6}$).

For both RSs, circular bundleviews as interactive tool can be applied to conduct what-if analyses on software-system structures [22] ($\underline{A1}$, $\underline{A4}$, $\underline{A6}$). They can also be used during collaborative decision processes involving multiple developers to decide upon multiple refactoring options by virtually *editing* such structures and supporting visual exploration of implied change impact ($\underline{A5}$) and development activities ($\underline{A2}$).

In a similar manner, a technique for comparing multi-dimensional relationships between nodes uses linearized two-dimensional plots [23]. In such plot, each dimension of links is 'rolled out' next to previous ones (Fig. 4). Nodes and their hierarchy are always aligned along the vertical axis ($\underline{A1}$), links along the horizontal axis. By that, correlations between analyzed relationship dimensions can be identified by similar or repeating patterns on the horizontal axis ($\underline{A3}$): For instance, this can be used to concurrently analyze selected relationships of modules before and after a proposed refactoring ($\underline{A5}$).

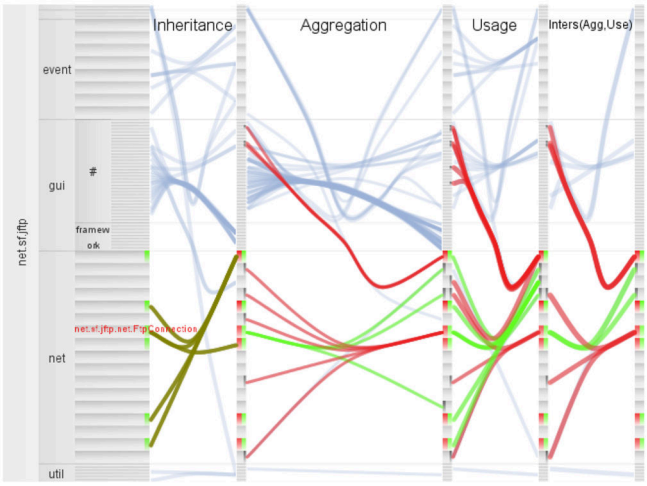

Figure 4. A software map depicting multi-dimensional relationships between entities (courtesy of Beck et al. [23]).

Table II
FITNESS FOR PURPOSE OF DISCUSSED SOFTWARE MAPS.

Technique	Compre-hensibility	Strategic	Technical	Addressed needs
Treemap	●●●	●●●		M2, M4, M5, A3, A2
Circ. Bundleview	●●●	●●●	●	M2, M4, M5, A2, A4, A5, A6
Multi-Dim Plot	●●	●	●●	A1, A3, A5
Call Graph	●	●	●●	Q3, Q4
Dep. Viewer	●		●●●	Q2, Q3
Call Timeline	●		●●●	Q3, Q4

D. Software Maps: Fitness for Purpose

Different software map techniques fit different levels of abstraction, tasks and communication (Table II). Treemap and circular bundleview can be used for high-level, holistic depiction of complex structures. So they serve well as communication means for various perspectives in software engineering processes and can also be understood by non-technical staff for strategic purposes. The bundleview's edge-grouping technique further reduces complexity when displaying large sets of relationships. On the contrary, high-detail depictions, such as UML graphs and call timelines, focus on specific properties of software-engineering data. Thus, they work particularly well for in-depth analysis of that data. But in turn, they are typically less intuitive for non-technical staff and mostly work for discussions among technical staff.

IV. Related Work

Teyseyre and Campo [24] provide a broad overview of three-dimensional tools and techniques for software visualization and examine potential application domains of these techniques. Their evaluation focuses on three-dimensional techniques, leaving aside important two-dimensional one.

Roman and Cox [25], Petre et al. [26], and Maletic et al. [27] categorize a number of software visualization tools by several criteria, including supported tasks and audience. In contrast, they do not evaluate how the tools can be useful with respect to both the important collaborative aspects as well as decision-making processes inherent to software engineering projects.

V. Conclusions

We have presented an overview of perspectives on software engineering processes that we observed during visits to industry partners. For each perspective, we identified communication and information needs, derived from its responsibility and typical communication links. We subsequently examined a number of existing recommendation systems that support activities performed by personnel representing these perspectives. Identified decision-making processes were then used to discuss how these processes can be further assisted by means of different software maps.

A key benefit of this extension to recommendation systems is that software maps provide a complementary picture of underlying software engineering data and processes. Besides our arguments, the real usefulness of software maps as extensions to recommendation systems yet remains to be proven by installing them in software engineering projects and conducting empirical validation. Nevertheless, we feel confident that their benefits will convince future users.

References

[1] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Softw.*, vol. 27, pp. 80–86, 2010.

[2] G. C. Murphy and E. Murphy-Hill, "What is trust in a recommender for software development?" in *Proc. RSSE*. ACM, 2010, pp. 57–58.

[3] J. Bohnet and J. Döllner, "Monitoring code quality and development activity by software maps," in *Proc. MTD*. IEEE, 2011, pp. 9–16.

[4] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proc. ICSE*. IEEE, 2007, pp. 344–353.

[5] P. Kruchten, *The Rational Unified Process: An Introduction*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[6] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proc. ICSE*, 2006, pp. 492–501.

[7] S. Henry and S. Wake, "Predicting maintainability with software quality metrics," *J SOFTW MAINT RE-PR*, vol. 3, no. 3, pp. 129–143, 1991.

[8] J. H. Hayes, S. C. Patel, and L. Zhao, "A metrics-based software maintenance effort model," in *Proc. CSMR*. IEEE, 2004, pp. 254–258.

[9] T. A. B. Pereira, V. S. dos Santos, B. L. Ribeiro, and G. Elias, "A recommendation framework for allocating global software teams in software product line projects," in *Proc. RSSE*. ACM, 2010, pp. 36–40.

[10] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *IEEE TVCG*, vol. 12, pp. 741–748, 2006.

[11] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol, "Not all classes are created equal: toward a recommendation system for focusing testing," in *Proc. RSSE*. ACM, 2008, pp. 6–10.

[12] M. Wilhelm and S. Diehl, "Dependency viewer - a tool for visualizing package design quality metrics," in *Proc. VISSOFT*. IEEE, 2005, pp. 34–35.

[13] C. McMillan, D. Poshyvanyk, and M. Grechanik, "Recommending source code examples via api call usages and documentation," in *Proc. RSSE*. ACM, 2010, pp. 21–25.

[14] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal api rules from imperfect traces," in *Proc. ICSE*. ACM, 2006, pp. 282–291.

[15] T. Xie and J. Pei, "Mapo: mining api usages from open source repositories," in *Proc. MSR*. ACM, 2006, pp. 54–57.

[16] B. Fluri, J. Zuberbühler, and H. C. Gall, "Recommending method invocation context changes," in *Proc. RSSE*. ACM, 2008, pp. 1–5.

[17] J. Bohnet, S. Voigt, and J. Döllner, "Locating and understanding features of complex software systems by synchronizing time-, collaboration- and code-focused views on execution traces," in *Proc. ICPC*. IEEE, 2008, pp. 268–271.

[18] J. Trümper, J. Bohnet, and J. Döllner, "Understanding complex multithreaded software systems by using trace visualization," in *Proc. SOFTVIS*. ACM, 2010, pp. 133–142.

[19] O. Hummel, W. Janjic, and C. Atkinson, "Proposing software design recommendations based on component interface intersecting," in *Proc. RSSE*. ACM, 2010, pp. 64–68.

[20] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proc. SOFTVIS*. ACM, 2010, pp. 5–14.

[21] N. Tsantalis and A. Chatzigeorgiou, "Ranking refactoring suggestions based on historical volatility," in *Proc. CSMR*. IEEE, 2011, pp. 25–34.

[22] M. Beck, J. Trümper, and J. Döllner, "A visual analysis and design tool for planning software reengineerings," in *Proc. VISSOFT*. IEEE, 2011, pp. 54–61.

[23] F. Beck, R. Petkov, and S. Diehl, "Visually exploring multidimensional code couplings," in *Proc. VISSOFT*. IEEE, Sep. 2011, pp. 1–8.

[24] A. R. Teyseyre and M. R. Campo, "An overview of 3d software visualization," *IEEE TVCG*, vol. 15, pp. 87–105, 2009.

[25] G.-C. Roman and K. C. Cox, "A taxonomy of program visualization systems," *IEEE Computer*, vol. 26, no. 12, pp. 11–24, 1993.

[26] M. Petre, A. F. Blackwell, and T. R. Green, *Software Visualization - Programming as a Multimedia Experience*. MIT Press, 1998, ch. Cognitive Questions in Software Visualisation, pp. 453–480.

[27] J. I. Maletic, A. Marcus, and M. L. Collard, "A task oriented view of software visualization," in *Proc. VISSOFT*. IEEE, 2002, pp. 32–40.